The MIPS R4000, part 8: Control transfer

devblogs.microsoft.com/oldnewthing/20180411-00

April 11, 2018

Raymond Chen

Let's just get this out of the way.

The MIPS R4000 has branch delay slots.

Ugh.

When you perform a branch instruction, the instruction after the branch instruction is executed, *even if the branch is taken*. The branch itself is delayed by one instruction.

This takes a lot of getting used to.

And to make things even more confusing, there are situations where the instruction in the branch delay slot is ignored, But lets not get into that yet.

Here's a basic example of a branch delay slot:

```
    BEQ     r1, zero, dest      ; branch if r1 is equal to zero
    OR      r1, zero, zero      ; set r1 = 0; this line executes regardless
    ...
dest:
    ADDI    r2, zero, 1         ; set r2 = 1
```

The `OR` instruction sits in the branch delay slot, and it will execute regardless of whether the branch is taken. It still executes *after* the branch instruction, so don't think of the entire branch instruction as executing after its delay slot. Only the control transfer part executes late. (In the above example, the `BEQ` tests the previous value of *r1*, not the value set by the `OR` instruction that sits in the delay slot.)

In the above example, the sequence of events is as follows, with time proceeding to the right.

Fetch and decode `BEQ` instruction.	→	The condition is true, so fetch the next instruction from `dest`.

	Fetch and decode `OR` instruction.	→	Set *r1* to zero.
			Fetch and decode `ADDI` instruction. → Set *r2* to 1.

When the branch instruction executes, the fetch and decode of the instruction in the branch delay slot is already under way. Instead of throwing away that work, the processor says, "Well, I may as well finish what I've started, seeing as I've already paid for it," and it polishes off its drink before leaving the table to fetch and decode the instruction at the branch destination. The result is that the control transfer doesn't happen until one straggler instruction has already executed.

Basically, this is a trick to avoid a pipeline bubble during branching without needing complicated speculation circuitry. On a system without branch delay slots, the processor has a lot of decisions to make the moment it decodes a branch instruction.

- A branch predictor decides whether instruction fetching and decoding continues from the branch-taken or the branch-not-taken code path.
- Speculation circuitry executes the operations from the predicted code path, but doesn't commit the results. Any side effects (such as register updates, memory updates, and exceptions) must be suppressed until the processor determines whether the branch is taken. If speculation was correct, than all side effects are realized. If speculation was incorrect, than all side effects are discarded.

Branch delay slots remove all this complexity. The processor always fetches and executes the straight-line code. And the processor has determined whether the branch is taken by the time it needs to fetch an instruction beyond the the branch delay slot. This means that you don't need a branch predictor, return address predictor, or speculation, thereby reducing chip complexity.

Unfortunately, branch delay slots also expose the internal pipelining. If a future version of the processor has a different pipeline depth, it still needs to emulate the old pipeline timing.[1]

Okay, before we get even more bogged down in the intricacies of branch delay slots, let's look at the control transfer instructions. First, the conditional transfers:

```
; all comparisons are signed
BEQ     rs, rt, dest    ; branch if rs = rt
BNE     rs, rt, dest    ; branch if rs ≠ rt
BGEZ    rs, dest        ; branch if rs ≥ 0
BGTZ    rs, dest        ; branch if rs > 0
BLEZ    rs, dest        ; branch if rs ≤ 0
BLTZ    rs, dest        ; branch if rs < 0
```

The branch instructions have a reach of ±128KB.

To help prepare for one of the above branch instructions, you can use one of these instructions:

```
; "set if less than"
SLT     rd, rs, rt      ; rd = (( int32_t)rs < ( int32_t)rt)    ? 1 : 0
SLTI    rd, rs, imm16   ; rd = (( int32_t)rs < ( int16_t)imm16) ? 1 : 0
SLTU    rd, rs, rt      ; rd = ((uint32_t)rs < (uint32_t)rt)    ? 1 : 0
SLTIU   rd, rs, imm16   ; rd = ((uint32_t)rs < (uint32_t)(int16_t)imm16) ? 1 : 0
```

The `SLT` family of instructions compare two values and set the destination register to 1 if the first comparand is less than the second; otherwise it sets the destination register to 0. The `I` versions compare against a signed 16-bit immediate value rather than a register, and the `U` versions use an unsigned comparison instead of signed. Note that `SLTIU` sign-extends the immediate from a 16-bit value to a 32-bit value, but the comparison is performed as an unsigned value. No arithmetic exceptions are raised by these instructions.

The assembler provides several pseudo-instructions for other types of relative branches, often using one of the `SLT` instructions to build the branch condition in the *at* register. Here are a few of them:

```
BEQ  zero, zero, dest   ; B dest
                        ; branch unconditional

BEQ  rs, zero, dest     ; BEQZ rs, dest
                        ; branch if rs = 0

BNE  rs, zero, dest     ; BNEZ rs, dest
                        ; branch if rs ≠ 0

LI   at, imm32          ; BEQ rs, imm32, dest
BEQ  rs, at, dest       ; branch if rs = imm32

LI   at, imm32          ; BNE rs, imm32, dest
BNE  rs, at, dest       ; branch if rs ≠ imm32

SLT  at, rs, rt         ; BLT rs, rt, dest
BNEZ at, dest           ; branch if rs < rt

SLTI at, rs, imm16      ; BLT rs, imm16, dest
BNEZ at, dest           ; branch if rs < imm16

LI   at, imm32          ; BLT rs, imm32, dest
SLT  at, rs, at         ;
BNEZ at, dest           ; branch if rs < imm32

SLT  at, rs, rt         ; BGE rs, rt, dest
BEQZ at, dest           ; branch if rs ≥ rt

SLTU at, rs, rt         ; BGEU rs, rt, dest
BEQZ at, dest           ; branch if rs ≥ rt (unsigned)
```

And so on. You get the idea.

The next batch of control transfer instructions are those which perform relative branches and store the return address in the *ra* register. The return address is the instruction after the branch delay slot.

```
; all comparisons are signed
BGEZAL  rs, dest        ; branch if rs ≥ 0 and link
BLTZAL  rs, dest        ; branch if rs < 0 and link

; pseudo-instruction
BGEZAL  zero, dest      ; BAL dest
                        ; branch unconditional and link
```

The `BAL` pseudo-instruction is an unconditional branch and link, implemented by encoding a conditional branch and link where the condition is always true (namely $0 \geq 0$).

The next batch is the "likely" branch instructions. These not only include a hint that the branch should be predicted taken, but they also have the extra weirdness that the instruction in the branch delay slot is *ignored* if the branch is not taken!

Note carefully: The instruction in the branch delay slot is ignored if the branch is *not taken*. If the branch is taken, then the instruction in the branch delay slot executes normally.[2]

```
; all comparisons are signed
BEQL    rs, dest        ; branch if rs = 0, likely
BNEL    rs, dest        ; branch if rs ≠ 0, likely
BGEZL   rs, dest        ; branch if rs ≥ 0, likely
BGTZL   rs, dest        ; branch if rs > 0, likely
BLEZL   rs, dest        ; branch if rs ≤ 0, likely
BLTZL   rs, dest        ; branch if rs < 0, likely

BGEZALL rs, dest        ; branch if rs ≥ 0 and link, likely
BLTZALL rs, dest        ; branch if rs < 0 and link, likely
```

The MIPS people presumably reconsidered these instructions, because later versions of the architecture mark them as deprecated.

The last group are the "jump" instructions.

```
J       dest            ; jump
JAL     dest            ; jump and link
                        ; return address stored in ra

JR      rs              ; jump register
JALR    rd, rs          ; jump and link register
                        ; return address stored in rd
```

The first two instructions encode an absolute jump, sort of. The `J` and `JAL` instructions have room to express the lower 28 bits of the absolute jump target. The upper four bits of the jump target are copied from the program counter of the branch delay slot. (This is almost always the same as the program counter of the jump instruction itself; it makes a difference only if the jump instruction and the branch delay slot are on opposite sides of a 256MB boundary.)

For example, suppose the program counter is at `0x12345678`. This means that the jump instruction can jump to any address in the range `0x10000000` through `0x1FFFFFFC`.

This partitioning of the jump target space into 256MB regions means that in practice, a DLL cannot exceed 256MB, and a DLL cannot be relocated so that it straddles a 256MB boundary, because it would be impossible to fix up the jumps that cross the boundary.

The jump register instructions use a register to specify a jump target. Unlike the absolute jump instructions, the jump register instructions can jump to any 32-bit address.

The `JALR` instruction is the only control transfer instruction that lets you pick the register to receive the return address. In practice, you always pick *ra*, but the possibility is nevertheless available to pick something else, in case you're doing something wacky.

Okay, now back to branch delay slots.

One rule about branch delay slots is that you cannot put another branch instruction in a branch delay slot. Because that would be *Inception*-level crazy.

Another rule about branch delay slots is that if an exception occurs while executing the instruction in the branch delay slot, and the kernel decides to resume execution after fixing the problem, execution will resume *at the preceding branch instruction*.

This is obvious in retrospect, because if execution resumed at the branch delay slot, well, there's no branch instruction active when execution resumes, so execution will fall through, which is bad if the original exception had occurred when executing the instruction in the branch delay slot for a taken branch. Resuming from the instruction that raised the exception would cause the taken branch to become not-taken!

Therefore, the kernel backs up to the branch instruction and resumes execution there. Branch instructions cannot fault, and they modify at most *ra*; in particular, the register being tested by a conditional branch did not change, so the resumed execution will take or not-take the branch in the same way as the original execution, and the instruction in the branch delay slot will get another chance to execute.

Well, I sort of lied when I said that "the register being tested by a conditional branch did not change": If the register being tested *is ra itself*, then the branch instruction will indeed modify the register that controls the conditional branch! (Similarly if you write `JALR r, r`.)

So let's just say that it's not allowed by convention. A branch instruction cannot be conditional on *ra* if it also modifies *ra*, and `JALR r, r` is also not allowed by convention.[3]

These forbidden instructions are merely software conventions. The processor will gladly let you do these disallowed things, but if you try it, and you take an exception on the instruction in the branch delay slot, then your program will act all wacky when execution is resumed, and you got what you deserved.

Oh yeah, you also shouldn't put a multi-instruction pseudo-instruction in the branch delay slot, because only the first instruction will execute as part of the branch delay slot. I never tried it, but I hope the assembler warns you when this happens.

Branch delay slots are a place you are likely to see `NOP` instructions. If the compiler can't find anything to put in the branch delay slot, it will just dump a boring `NOP` in there.

Next time, we'll look at some of the crazy things you can do with branch delay slots. I don't know if any compilers take advantage of them, but they are technically legal.

[1] I'm led to believe that this problem actually occurred. The original MIPS processor was single-issue with a two-stage pipeline. Later versions deepened the pipeline and added multi-issue, which means that a single branch delay slot is not sufficient to avoid a pipeline bubble. So they had to add branch prediction circuitry anyway.

[2] The story as I heard it is that the MIPS folks noticed that lots of people were putting `NOP` instructions in the branch delay slot because they couldn't find anything useful to go in there. So the MIPS people added the "likely" version of the branch instructions which allows you to front-load the first instruction of the jump target in the branch delay slot. If the branch is not taken, the instruction in the branch delay slot is ignored, which means that you don't have to worry about the front-loaded instruction interfering with the fall-through code path. Though they ended up deprecating the instructions, so who knows what really happened. Maybe an intrepid reader can dig up a design document.

[3] Furthermore, the Windows NT calling convention requires that the return address be passed in *ra*, so you can cancel your dreams of using `JALR` on Windows NT with a nonstandard return address register.

Bonus chatter: Intrepid reader laonianren dug up some usenet articles that fill in the background.

The branch-likely instructions were added "only to make it easier to populate the branch delay slots in loops": You can put the first instruction of the loop body in the delay slot, and branch to the *second* instruction of the loop body. If the branch is not taken, then the instruction in the branch delay slot is converted to a NOP. (It still consumes a cycle, but nothing happens during that cycle.)

The branch-likely instructions were deprecated because "conditionally squashing the result of the delay-slot instruction is a pain in the neck." Processors are required to implement the instructions, but they are not required to implement them *efficiently*.

Raymond Chen

Follow